PGR3302

Software Design

Mappeeksamen i gruppe

Monopol

Høst 2021

PGR3302 Software Design Eksamen

Introduksjon:

Programmet kan kjøres fra en .exe-fil. I introduksjonen blir brukerne bedt om å taste inn antall spillere (minimum to og maks fire), og deretter må de skrive inn navnet sitt. Dersom brukerne ikke skriver det inn, settes det en default verdi automatisk (*easter egg*). Alle spillerne starter fra samme posisjon, og kaster terning etter tur. Hver gang en spiller passerer START, får de 100M som bonus. Dersom en spiller lander på fengsel, må vedkommende hoppe over en runde. Spillet fortsetter til én spiller gjenstår.

Prosess:

Ved utvikling av prosjekteksamen i PGR3302 Software Design startet vi prosessen med å komme med ulike forslag til hva vi kunne utvikle. Vi skrev ned alle ideene vi hadde på et tankekart, både gode og dårlige. Dette gjorde vi fordelt over 3 faser, hvor vi diskuterte de forskjellige forslagene dypere for hver fase, og argumenterte for og imot. I den siste fasen satt vi igjen med tre ulike ideer - en reiseplanlegger, et drikkespill og Monopol i konsoll. Vi lagde klassediagram for to av dem for å vurdere hvilke som var mest realistisk i forhold til tidsrammen og gjennomføringen. Vi tenkte at drikkespillet ble for enkelt, så da sto det mellom de to siste. Valget landet enstemmig på Monopol i konsoll, da dette virket mest interessant, samt utfordrende å utvikle.

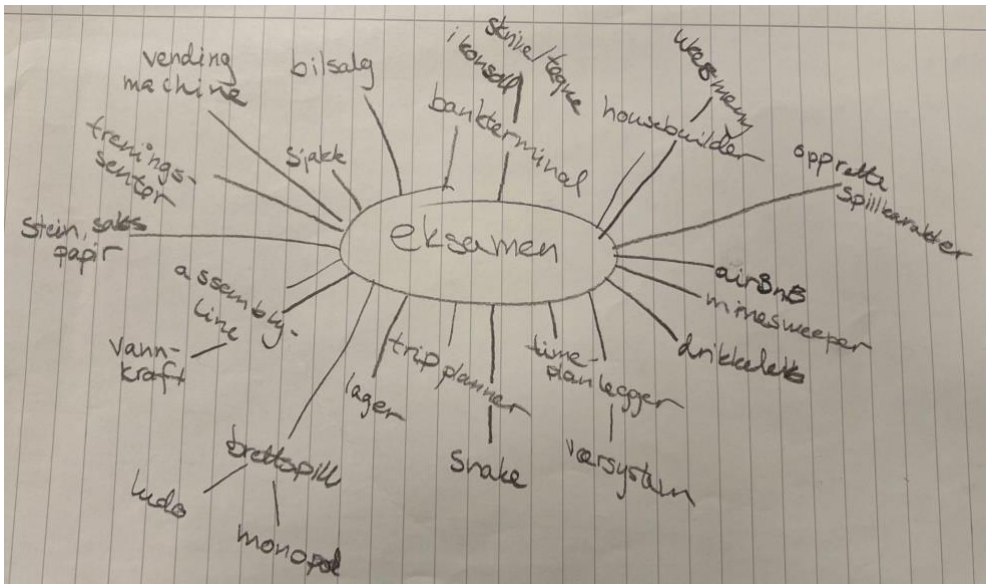
Samtidig som vi idémyldret hva vi skulle lage, prøvde vi å komme med forslag til hvilke patterns vi kunne bruke for å løse dem. Da valget falt på Monopol fullførte vi resten av klassediagrammet vi hadde begynt på. Vi fant ut hvilke patterns vi skulle bruke, og hvilke relasjoner de hadde til hverandre. Videre lagde vi et implementation class diagram hvor vi gikk grundigere over hvilke metoder vi skulle implementere i de ulike klassene. Vi ble enige om å planlegge grundig før vi skulle begynne på kodingen, og la dermed av en del tid til dette i begynnelsen.

Da vi startet på kodingen delte vi oss inn i to og to, hvor vi valgte en del av programmet for å kode sammen på. Vi brukte teknikker som parprogrammering og testdrevet utvikling (TDD) for å komme i gang. Deretter delte vi oss opp litt mer, og jobbet på ulike brancher med de forskjellige delene av programmet. Underveis ved implementering av klassene, fant vi ut at vi måtte endre litt på dem i forhold til hva vi i utgangspunktet hadde planlagt. Ettersom vi hadde en ganske grundig planlegging, var det ikke mye som måtte endres. Da vi nærmet oss med å bli ferdig med de ulike klassene, koblet vi sammen de som skulle kalle på hverandre, og testet at de fungerte slik de skulle. Vi jobbet oss inn mot GameManager klassen, slik at MenuUI lett kan få tilgang til logikken derifra. Vi testet så spillet fra console, og rettet opp i bugs som ble oppdaget. Vi la også inn ekstra funksjonaliteter i programmet som vi anså som nødvendig for brukeropplevelsen.

Gjennom hele prosjekteksamen hadde vi et skjema med hvilke dager og klokkeslett vi skulle jobbe, samt en agenda og mål for hva vi skulle gjøre hver dag. Vi jobbet fysisk, da vi synes dette fungerte best for oss. Underveis har vi lagt inn kommentarer på rapporten for å sørge for at vi har fått med oss alt vi ønsket og eventuelle bugs.

Idéer:

Fase I: (Brainstorming)



Fase II: (3 eller flere stemmer går videre til fase III)

- **III** forhold til WPF:
    - En visuell bankterminal. Første gang man starter opp må man opprette en kode. Etter dette må man skrive inn denne koden for å betale. *Database, logging.*
- **IIII** Trip planner (Decorator). Legge til destinasjoner fra start til slutt
    - Velge transportmåte, får brukt factory DP
- Restaurant - bygge en egen pizza, velge leveringsalternativ
- **III** Timeplanlegger / Kalender
- **III** House builder. Bygg ditt eget hjem med decorator-pattern. Har huset stue?
- **IIII** Brettspill (f.eks. LUDO, Monopol) + terning
- **IIII** Drikkelek / Kahoot type spill

Fase III: (Mer utdypning)

- **Trip planner. Legge til destinasjoner fra start til slutt**
    - Velge transportmåte, får brukt factory DP
    - DB: Destinasjoner
    - Decorator DP -> Legge til ekstra destinasjoner etter start.
    - Factory DP -> Ved valg av reisemåte (transportmiddel)
    - Hvis tid: Timeplanlegger / Kalender -> Velg avreisedato. Avstanden mellom de forskjellige destinasjonene vil summere en ETA.
- **Monopol**
    - Factory → Henter korttype (Eiendom, fengsel, start, prøv lykken)
    - Singleton → Logger (spillerposisjon, kroner, eiendomsporteføljе)
    - Flyweight -> Spiller faste verdier (id, navn), ikke fast (valuta)
    - Facade -> Kan kanskje samle flere ofte brukte metoder i en klasse.
- **Drikkelek / Kahoot type spill**
    - Database med “premade” spørsmål + en hvor bruker kan legge inn spørsmål
    - Spill enten med premade pakker, premade pakker + egne spørsmål, eller bare egne spørsmål
    - Factory DP -> Et interface (statement) med to subklasser
    - Singleton DP -> logger
    - Decorator DP → “Kings Cup”

Spesielle utfordringer/ fancy features:

Design patterns

Vi brukte *factory design pattern* for å generere felt til monopolbrettet. Dette mente vi var en god løsning ettersom vi hadde behov for flere kort med forskjellige attributter, men som alle implementerer en samme metode for å printe ut kortverdier. Det var veldig behagelig å benytte factory DP ettersom det ga oss friheten til å velge hvilke korttyper vi ønsker å ha på brettet uten å endre koden i selve factory DP. Det å overlevere ansvaret for opprettelsen av nye kort til en egen klasse, gjorde også at det var veldig enkelt å f.eks. legge til flere sjansekort.

Vi valgte å benytte oss av *flyweight design pattern* for å opprette alle spillere med deres attributter. Vi ønsket å vise at vi behersket flyweight, det virket som en god anledning ettersom alle spillere har både en unik del og en felles del når de opprettes. I "flyweight-factory" klassen lagret vi spiller-objektene og deres respektive id'er i et "Dictionary". I tillegg hadde vi en metode for å fjerne ting derfra. Utfordringen var at vi i flere klasser hadde behov for tilgang til samme instans av denne klassen for å ha tilgang til de samme spillerne. Løsningen sto da mellom å enten lage flyweight-factory static, eller å kombinere flyweight-factory klassen med *singleton*, eller å kutte ut flyweight fullstendig og finne en annen løsning for generering av spillere.

Vi valgte å beholde flyweight, men å kombinere det med singleton patternet slik at vi fikk tilgang riktig instans av PlayerGenerator-klassen. Grunnet behovet for våre modifikasjoner fra eksempelet vi bl.a. gikk gjennom i undervisningen, er det nok ikke det mest hensiktsmessige stedet å implementere flyweight. Men vi ville likevel implementere det slik at vi fikk vist frem at vi behersker patternet. Argumentet for å benytte static/singleton var at vi hadde flere klasser som er avhengige av informasjon i PlayerGenerator. I undervisningen har vi blitt opplært til å opptre skeptisk når det gjelder globalt tilgjengelige deler av koden. I dette tilfellet, når vi har et stadig behov for å både ha tilgang til samt å gjøre endringer på elementer i samme instans av en klasse, så fant vi ingen annen effektiv måte å gjøre dette på. Både det å gjøre klassen static og implementasjon av singleton DP vil gjøre informasjonen allment tilgjengelig, men det er etter vårt skjønn en ryddigere måte å gjøre det på via singleton pattern.

Grunnet behov for tilgang til GameManager sine attributter valgte vi å implementere singleton-pattern i klassen. Denne beslutningen fattet vi med grunnlag i at det var flere klasser som var avhengig av tilgang til samme instans av både listen av 'controllers' og kartet over brettet. Valget stod dermed mellom å enten gjøre klassene der vi oppretter listene og kartet globalt tilgjengelig, eller å kun gjøre dette med GameManager. På grunnlag av at vi ville holde minst mulig klasser globale, valgte vi å gjøre dette kun med GameManager. I tillegg valgte vi i begge implementasjoner av singleton; å legge til en lås slik at den er trådsikker. Vi benytter ikke multithreading i løsningen, men vi tenkte at det var lurt med tanke på videre utvikling.

JSON

Vi benytter en JSON fil for å holde på lagrede data/ verdier til sjansekort og eiendommer. I undervisningen ble både ADO.NET, JSON og XML nevnt som måter å få til dette på. Ettersom vi ikke skulle aktivt lagre mye i en database, og vi ikke gikk veldig nøye inn på hvordan sette opp en database i undervisningen, valgte vi å gå bort fra ADO.NET. Vi bestemte oss for å lage en JSON fil som vi henter opp informasjon fra. Å forstå hvordan å implementere JSON var utfordrende, særlig med tanke på at vi i undervisningen ikke har gått i dybden om hvordan dette gjøres. Det måtte en del google-søk til for å få satt opp, men da vi fikk det til var det veldig forståelig og lett å bruke.

Boardmap

Det som opplevdes utfordrende under utvikling av spillbrettet var at det skulle fremstilles visuelt i console, med ruter som hadde indeksverdier som gikk med klokken slik at spillerne kunne rokkere rundt brettet. Dette betydde at vi måtte ha stigende indeksverdier på venstre side, og synkende indeksverdier på høyre side, i tillegg til å måtte ta høyde for console output som printer ut linjevis. Dette løste vi først ved å printe ut to go to indeksverdier om gangen (en fra venstre og en fra høyre), og la inn en differanse mellom dem.

Bugs:

Vi har testet spillet/ koden flere ganger, og har fikset alle bugs vi har oppdaget. Per nå har vi ingen bugs som vi er kjent med.

Kildehenvisning:

Vi har ikke benyttet materiell utover undervisnings slidene.

Liste over elementer fra pensum:

- Design patterns: Factory, Flyweight og Singleton.
- Versjonskontroll (GitHub/ Git)
- Properties
- Parprogrammering
- Abstract class og interface
- StringBuilder
- Region
- Serialisering Json file
- Debugging
- Casting
- Try/Parse
- SOLID - Hovedfokus på S, O og L
- Refactoring
- UML Domain Model
- Implementation Diagram
- Uml sekvensdiagram
- Unit testing (TDD)
- Operator overloading
- Layers
- Virtual